**关于异常**

## 异常与错误码

异常与错误码, 无论是以前还是现在,我相信大家在这两者之前在选择使用哪一个上都可能会有一定的疑惑.虽然我通过阅读知道应该使用异常来报告错误,而不应该是错误码.在我的上一家公司却不是这样选择的.我也努力去反应这个问题,却因各种理由(包括历史原因)最终依然选择错误码来报告错误.最终我也必须使用错误码来报告错误.现在终于有机会回到了正确的道路上来,今天就来分享一下我对异常的认识.

至于为什么这样选择?在.NET方面的书籍上都会有各种理由,但是有时会让人无法做出抉择.让人的感觉就是异常也可以报告,错误码好像也行.为了更加坚定的选择异常.我从以下几方面总结了我们该选择异常,而不是错误码

### 统一性

* 异常与各种面向对象语言集成得非常好,而错误码就差很多.

在.NET中,有些地方是无法使用返回值的,比如在构造函数、操作符重载、转换操作符以及属性中,对返回值没有什么选择的余地.所以在程序运行错误时就没有办法使用错误码来报告错误.那么为了整个系统错误报告的统一性.异常就是唯一的选择了.

* 异常增强了API的统一性.

设计异常的目的就是用来报告错误的,而返回值的使用有多种用途.另外在.NET框架中都是使用异常来报告错误,如果在自己的应用程序中使用了错误码来报告异常,那么在应用程序中就应有两种方式对错误做出处理,显然两种方式增加了系统的复杂度.降低了系统的可维护性.所以两种方式还不如一种方式.

### 稳定性/安全性

* 错误码容易被忽略,而无法忽略通过异常报告的错误,从而使代码变得更加稳固、安全

错误码代表什么?如果没有一个详细的错误码文档.我们很难对-1等返回值具有什么意义有一个了解,从而对编程带来不可确定性.并且容易产生错误.这个稍后会有例子说明.

另外基于返回值的错误处理时,如果程序调用失败,那么程序会在结果不正确的情况下继续运行,并将导致程序崩溃或数据被破坏.而如果使用异常来报告错误,一旦发生错误,线程会被暂停,调用代码则需要对异常进行处理.如果没有处理异常,应用程序会被终止.终止应用程序比让它继续不正常的运行要好,错误总是会被修正的.

* 错误码无法应对意料之外的错误,程序会继续运行并可能导致未定义的结果.如果使用了异常,在发生意外的时候,应用程序可以调用未处理异常的处理程序异常进行处理或者关闭应用程序.

### 及时性

* 异常可以包含丰富的信息来对错误进行描述,有利于及时修复错误.而错误码其表示意义很难让人理解
* 异常易于检测、发现.

异常是专门用来报告错误的,在开发人员工具中(比如调试器)是时刻关注异常的发生.如果调试过程中出现了错误,则调试器会立刻中断.包括在另外一个线程中出现的错误.有利于在开发过程中及时的发现问题并修正问题.如果使用错误码来报告错误那么就得不到这些工具的支持.应用程序将会暗藏着很多问题

### 小结

其实个人理解是,异常的目的就是报告错误,让应用程序在开发过程上更好的去发现错误并修复, 避免程序在发布后存在大量错误.即使在程序发布后出现错误,异常也有助于解决问题

异常更多的是实实在在的发现错误并让开发人员尽快解决问题.而错误码则更多的表现为系统一切运行正常.其实并非如此.

所以基于以上理由我的选择是----使用异常来报告错误

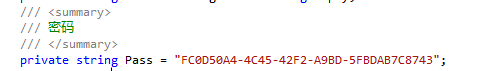
### 示例

下面摘自以前公司的数据库操作类





刚进公司的时候操作数据库类时使用到这个方法,当时没有查看代码的权限.一直执行失败,并返回-2,一直百思不得其解,执行ExcuteNonQuery在什么情况下会返回-2.从来没有这样的经验.确保了自己的SQL语句在查询分析器中是可以正常运行的.原后打开跟踪器,也一直没有看到语句的运行,当时一个纳闷啊,后来才知道





对于数据库连接只用配置服务器IP,以及数据源即可.但是数据库必须具有LibSvr以及密码为上图的一个用户实例. 虽然能够知道连接数据库失败并及时求助其它同事,才得已解决这个问题,但是我想这依然给我带来了很多困扰,看着上面蛋疼注释。谁知道我的程序出了什么问题.对于不具备在这样数据操作经验的人员来讲,如果不考虑数据库事务,很有可能直接写了ExcuteNonQuery之后并做了其它的数据库操作.如果上述代码不使用错误码,并且不捕获所有异常并吞掉异常,开发人员在使用过程中出现了问题会马上知道出现问题的原因并加以修复

* 这些都是由错误码没有异常报告错误能力的强大给开发人员带来的困惑。

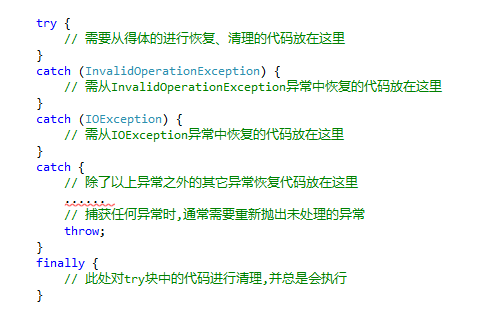
当我反应这个问题的时候,得到了两个理由

1. 互联网项目不应该将让用户看到系统出现了异常

2. 异常的使用会使系统性能低下

* 很多人担心异常的效率,但是在这里同样使用了异常(并不是正确的用法),最终确通过返回值来确定程序是否运行正常.完全不理解这种行为是出于什么目的.
* 另外这种多此一举的做法不仅对开发人员来讲带来了困惑,对程序也是一种糟糕的表现

## 异常处理机制



### try块

* Try块中的代码通常需要执行一些资源清理、或需要从异常中恢复
* 一个try块可以包含多个catch块

### catch块

* 包含响应一个异常需要执行的代码
* CLR是自上而下搜索匹配的catch块
* catch块的未尾,通常会有三个选择
  + 重新抛出相同的异常,向调用栈的更高层的代码通知异常的发生
  + 抛出一个不同的异常,向调用栈的更高层的代码提供更丰富的异常信息
  + 让线程从catch块的底部退出

### finally块

* 包含的代码是保证会被执行的,通常用于资源的清理工作

### 小结

避免在catch块与finally块中再次抛出异常

### 示例

异常处理不合理的写法



* 首先不管捕获所有异常,就资源的释放也应该写在finally中.如果上例中的代码开发人员是出于严谨考虑(显然没有达到目的),那么这个地方却又显得太不严谨了.

## 抛出异常及异常类型

实现自己的方法时,如果方法无法完成方法名所指明的动作时,就应抛出一个异常.抛出异常需要考虑两个问题

* 应该抛出什么类型的异常.

应该抛出一个有意义的异常,要考虑调用栈中更高层的代码,以便让它们能够执行一些更得体的异常恢复.抛出什么异常有两个原则,首先考虑FCL中是否包含与你需要表达的意思完全匹配的异常,有的话则应该首先使用FCL中已定义好的异常类型,否则可以定义算定义类型异常

* 抛出异常的消息

### 小结

* 不要让公有成员根据某个选项来决定是否抛出异常
* 不要把异常作为返回值返回
* 不要为了报告使用错误创建新的异常类型.
* 错误通常可以分为两种使用错误以及执行错误
* 使用错误说明程序写得不正确,可以通过修改代码来加以修复.不应该在程序中处理使用错误,而应该修改调用方的代码.比如因传入的参数为null而使程序进入错误状态.此时程序中会通过抛出ArgumentNullException来报告这种错误,那么调用方就应该保证绝对不会在参数中传入null.
* 执行错误是那些无法通过编写”更好的”代码来加以避免的.执行错误又可以分为程序错误和系统失败
* 程序错误是那些可以在程序中处理的执行错误 ,比如File.Open抛出FileNotFoundException时,调用代码可以捕获异常,并创建文件
* 系统错误是指那些无法在程序中处理的执行错误,比如OutOfMemeryException
* 抛出System.Exception几乎总是错的

## 异常消息

发生异常了,那么怎么样让开发人员知道异常发生了什么异常(异常的类型)以及异常发生的具体原因以及该如何去避免异常.那么异常消息的合理设计将至关重要

* 异常消息面向的对象

对于开发人员来讲,异常则要提供丰富而有意义的错误消息.让开发人员知道如何避免及修复;对于系统的使用人员(用户)来讲,提供详细的错误原因则不太合理.

* 确保异常消息的语法正常以及避免在异常消息中泄露安全信息

### 小结

对已处理的异常,异常消息并没有什么用,只有当异常未被处理时它们才能发挥作用,因为应用程序会被终止,未处理的异常显示系统存在的缺陷,修改错误的方法则是通过错误信息修改源代码.因此异常消息的目的应该是帮助开发人员修正代码中的错误.而不是给最终用户看的.

## StackTrace/Throw e/Throw

### StackTrace

* 异常堆栈跟踪信息.描述异常发生前调用的方法.

一个异常抛出时,CLR会在内部记录Throw指令的位置(抛出的位置),一个Catch块捕捉到该异常时,CLR会记录异常的捕捉位置,在Catch块中访问被抛出的异常对象的StacKTrace属性时,负责实现该属性的代码会调用CLR内部的代码,并创建一个字符串来指出从异常抛出位置到异常捕捉位置的所有方法

抛出一个异常时,CLR会重置异常的起点,CLR只记录最新的异常对象的抛出位置

### Throw e

* 抛出一个新的异常.LR会重置异常的起点,从而会记录并从此位置开始的异常的堆栈跟踪信息

### Throw

* 重新抛出原来的异常,但是CLR并不会重置堆栈的起点.

### 小结

获取异常发生的准备位置将非常有利于对错误代码进行修复.

## 异常处理

在决定何时抛出异常,确定了异常的种类时,并为之定义了异常消息后,该如何对异常进行处理了.首先异常处理是指在catch块中来捕获某个特定类型的异常,并完全理解在catch块之后让程序继续运行意味着什么,那么我们说这种情况就是对异常进行处理.比如,在File.Open方法抛出一个FileNotFoundException异常,捕获该异常时可以使用一个默认的文件来实现程序的正常运行.

* 不要在框架的代码捕获类型不确定的异常时,把异常吞了

将一个线程中抛出的异常转移到另个一个线程,那么则是可以捕获类型不具体的异常.比如在异步编程时.这种情况显然并不是把异常吞了

* 只有在确实知道异常产生的原因并能对错误做出正确的处理时,可以捕获具体人异常
* 不要捕获不应该捕获的异常.应该允许异常沿着调用栈向上游传递.(这点其实与第二点是一样子的,如果捕获了不应该捕获的异常,实际上是让应用程序中的缺陷更难以发现)
* 要在捕获异常后重新抛出异常时使用空的throw语句
* 合理的封装异常, 并为异常指定内部异常.底层抛出的异常传递到高层没有什么意义时,对底层的异常进行封装使异常对于高层的用户来讲是有意义的.

比如在静态构造函数中由于除0导致的异常,系统会抛出一个类型初始化异常来报告错误,让调用者知道异常发生在构造函数中.比知道除0导致的异常更有意义

另外对于一些底层异常消息描述的一些概念对于高层调用者来讲可能是无法理解的,因此对问题产生的原因没有太大的帮助.因此合理的封装异常对系统的稳固是有帮助的

### 小结

* 不要随便捕获异常,而是应该在测试的过程中让缺陷都暴露出来并全部修正,从而使发布后的应用程序更加的稳固.
* 每编写一个异常处理程序,都有可能会削弱系统的稳固性.而且如果犯了错误通常很难找到.因此异常请勿难用

## 性能及最佳实践

### 性能

* 使用Tester-Doer模式、Try-Parse模式来避免因异常而引起的性能问题
* 错误总是可能发生,但是并不经常发生.所以正确的使用异常对于系统性能的影响将会变得更小

### 最佳实践

* 善用finally块

不管线程抛出什么类型的异常,finally块中的代码都会执行

* 使用lock语句时,锁会在finally块中释放
* 使用using语句时,会在finally块中调用对象的Dispose方法
* 使用foreach语句时,会在finally块中调用IEnumerator对象的Dispose方法
* 定义析构方法时,会在finally块中调用基类的Finalize方法
* 不要什么都捕捉
* 捕捉一个异常时表明你预见到该异常,理解它为什么发生,并知道如何处理它.一个办法是修改代码来捕捉一个特定的异常,要么重写代码排除会造成抛出异常的出错条件.另外在一个catch块中确实可以捕捉System.Exception并执行一些代码,只要在这个catch块的未尾重新抛出异常.千万不要捕捉异常并悄悄的吞噬它,否则应用程序都不知道已经出错了
* 另外可以在一个线程中捕捉一个异常,在另外一线程中重新抛出异常.样例:异步编程模型
* 得体的从异常中恢复
* 发生不可恢复的异常时回滚部分完成的操作-----维持状态
* 隐藏实现细节来维持契约

有时可能需要捕捉一个异常并重新抛出一个不同的异常,这样做的唯一原因是维系方法的”契约”,

有的时候,开发人员之所以捕捉一个异常并抛出一个新异常,目的是在异常中添加额外的数据或上下文,如果这是你唯一的目的,那么只需捕捉希望的异常类型,在异常对象的Data属性中添加数据,并重新抛出相同的异常对象

## 总结